LUSTRE

Manuel de référence du langage

• Les commentaires en ligne débutent avec `--` et vont jusqu'à la fin de la ligne. Les commentaires "parenthésés" vont de `(*` au premier `*)` qui suit. *On ne peut pas imbriquer les commentaires parenthésés.*

• Les identificateurs (ident) sont des chaînes de caractères qui débutent par une lettre ou un "_", et qui contiennent des lettres, des chiffres et des "_".

• Les notations entières sont des chaînes de chiffres décimaux (ex. `0`, `24` ...).

• Les notations réelles débutent par une notation entière, suivie (impérativement) par une partie décimale et/ou une partie exposant (ex. `0.0`, `3.14`, `1e-10`, `0.33E+3`).

• Méta-langage :

Les terminaux sont en gras, les non-terminaux en italique. Les parties entre [ ] sont optionnelles. Les parties entre { }$^+$ peuvent être répétées une ou plusieurs fois. Les parties entre { }* peuvent être répétées zéro ou plusieurs fois.

• Un fichier lustre (extension `.lus`) contient une séquence de définition d'opérateurs (ou nœuds) :

fichier-lustre ::= { *nœud* }$^+$

• Un nœud est défini avec son nom, ses paramètres d'entrée et de sortie, ses éventuelles variables locales, puis son corps, constitué d'un ensemble d'équations :

nœud ::= **node** *ident* **(** *var-decl* { **;** *var-decl* }* **)** **returns** **(** *var-decl* { **;** *var-decl* }* **)** **;**
[**var** { *var-decl* **;** }$^+$]
let
{ *équation* }$^+$
tel

• Les variables sont éventuellement déclarées avec une horloge, qui doit être une variable booléenne. De plus, les horloges des entrées/sorties doivent être aussi des entrées/sorties :

var-decl ::= *var-type-decl*
(*var-type-decl* **)** **when** *ident*

• Les variables sont toujours déclarées avec leur type :

var-type-decl ::= *ident* { **,** *ident* }* **:** *type*

• En lustre de base, on n'utilise que les types prédéfinis (booléen, entier et réel) :

type ::= **bool** | **int** | **real**

• Chaque identificateur de sortie et de variable locale doit être défini par une et une seule équation. La deuxième forme permet de définir une liste d'indentificateur avec une seule équation : c'est en particulier le seul moyen d'instancier un nœud ayant plusieurs sorties :

équation ::= *ident* **=** *exp* **;**
| [**(**] *ident* { **,** *ident* }$^+$ [**)**] **=** *nuplet-exp* **;**

• Les expressions simples sont construites avec des constantes, des références à des variables ou des instances de nœuds à résultat unique, qu'on combine avec des opérateurs prédéfinis unaire, binaire ou ternaire (N.B. le "si alors sinon" est le seul opérateur ternaire prédéfini) :

exp ::= *constante* | *ident* | **(** *exp* **)** | *ident* **(** *exp* { **,** *exp* }* **)**
| *op-unaire exp* | *exp op-binaire exp*
| **if** *exp* **then** *exp* **else** *exp*

• Les expressions de n-uplets sont des instances de nœud à plusieurs sorties ou des n-uplets d'expressions simples, que l'on peut combiner avec des opérateurs polymorphes :

nuplet-exp ::= `(` *exp* { `,` *exp* }* `)` | *ident* `(` *exp* { `,` *exp* }$^+$ `)`
| *nuplet-exp* `->` *nuplet-exp* | `pre` *nuplet-exp*
| *nuplet-exp* `when` *exp* | `current` *nuplet-exp*
| `if` *exp* `then` *nuplet-exp* `else` *nuplet-exp*

• Les expressions doivent être correctement typées ; on donne en commentaires les combinaisons de types acceptées pour chaque groupe d'opérateurs. Les notations entières et réelles (base 10) suivent les même conventions que celles du langage C.

constante	::=	`true` | `false`	bool
	|	*notation-entière*	int
	|	*notation-réelle*	real
op-unaire	::=	`pre` | `current`	type → type
	|	`not`	bool → bool
	|	`-`	int → int, ou real → real
	|	`int`	real → int
	|	`real`	int → real
op-binaire	::=	`->` | `=` | `<>`	type × type → type
	|	`when`	type × bool → type
	|	`and` | `or` | `xor` | `=>`	bool × bool → bool
	|	`>` | `<` | `>=` | `<=`	int × int → bool, ou real × real → bool
	|	`+` | `-` | `*` | `/`	int × int → int, ou real × real → real
	|	`div` | `mod`	int × int → int